An Object-Oriented Augmentation of the Euclideus Language

Justin Gagnon

August 11, 2017

The present manual describes the implementation of object orientation to the Euclideus programming language, which hitherto followed a purely procedural paradigm. Two varieties of object-orientation have been implemented: class-based, and prototype-based object orientation. This manual serves to elucidate the key operators, syntax elements and ideas pertaining to this particular augmentation.

CONTENTS

1 OVERVIEW

Unlike the procedural programming language paradigm, which (loosely speaking) is all about "doing things" (executing commands), the object-oriented (OO) paradigm centers on "having things" (containing data) and "having the ability to do things". In real terms, a purely procedural language will store items in memory as long as it is using them but these items are discarded when no longer needed. In contrast, the point of an object-oriented language is to allow for the containment and preservation of such items in a certain memory structure such that these items (including their visible properties) may be recalled later on.

The Euclideus language accommodates an OO paradigm by identifying *objects*, *functions* and *classes* as key structural elements, which will be detailed throughout this manual.

1.1 OBJECTS

Objects form the core of the Euclideus programming language. An object, as shown in Fig. 1.1, is an entity that contains a certain set of information, or data, and functions which it can perform. An object's information content is in the form of a set of other visible and invisible objects. The object's functions are very much like standard functions, except that they have the ability to modify the object's own state, i.e. any of its data members[1]. A Euclideus object is one of four kinds: *primitive*, *native*, *instance* or *mutant*.

1.1.1 THE PRIMITIVE

The primitive object is the simplest kind, it is not composed of other objects. A primitive is essentially any expression that evaluates to a complex value – this includes algebraic (including expressions that include indeterminate symbols like "`a`, `b` or `c`", or function signatures e.g. "`f(x,y,z)`" coupled with the set of recognized operators), boolean or numerical expressions. The primitive object contains no objects nor classes, but it does contain a function called `eval()`, which evaluates the `primitive`'s associated expression. As the most fundamental object type, a primitive object may be used as a building block to create a mutant.

1.1.2 THE NATIVE OBJECT

A native object is a data structure that is pre-defined in the Euclideus system. It contains a set of members and functions but is not associated to any class, and thus its members cannot be inherited. Moreover, a native object cannot become a mutant (see below) under any circumstances. So far, only one such type of object, the array, is provided, but future updates to the Euclideus system will include additional pre-defined structures such as strings, maps, lists, etc.

[1] In contrast, a standard function which does not belong to any object is unable to modify data outside of its own scope.

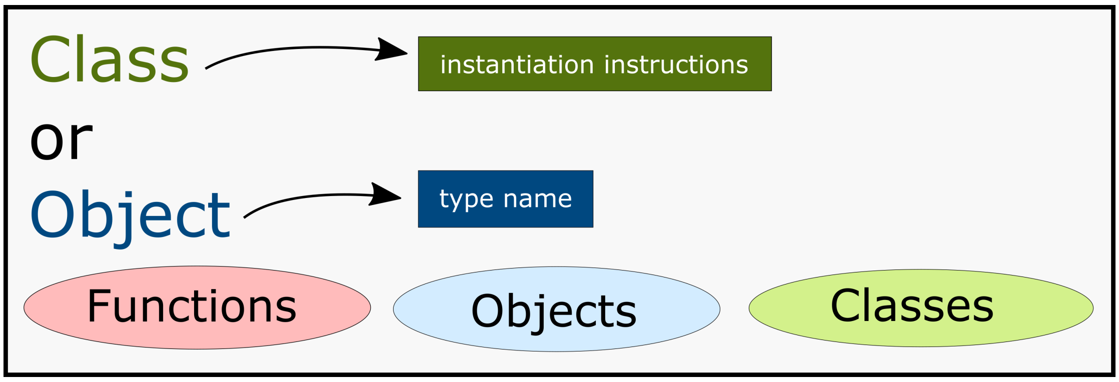


Figure 1.1: The basic structure of objects and classes. Classes and objects are very similar in terms of their contents. Both contain a set of function members (red bubble), object members (blue bubble) and class members (green bubble). A class contains additional information (dark green box) about how objects are instantiated from it, while objects contain the name of their class, i.e. their type name (dark blue box). This however doesn't mean that objects and classes are interchangeable because the difference lies in their mechanics: objects are instantiated from classes, but classes are never instantiated from objects (at least not in the current framework).

1.1.3 The Instance Object

An instance object is one which is constructed, using Euclideus' object-oriented syntax, from a well-defined recipe, called a *class*. Such an object is said to be of a certain *type*, its type being the name of the class that it is built (instantiated) from. An instance object has a well-defined structure – the instance object's members, functions and classes are all *immutable*, i.e. their type cannot be altered; and moreover its structure is *fixed* – members cannot be added to nor deleted from it.

1.1.4 The mutant Object

A mutant object is perhaps the diametric opposite of an instance (class-based) object. It is entirely mutable – its members are all mutable which means that they may be re-assigned to any other structure, they can also be removed and new members may be added. There are currently three ways in which to create a mutant: (i) a mutant is automatically generated when assigning a new field to a primitive object, (ii) the `mutate()` function allows to directly mutate an instance object, (iii) upon deletion or modification of a class, any instances of the class or any objects that contain members depending on the class which has been modified/deleted automatically become mutants.
To create any of the above quantities, an assignment statement is defined, using the "equals" "=" symbol. This symbol serves to assign classes, functions and objects.

1.2 Functions

Functions in Euclideus are either programmatically-defined – those which are created using the standard syntax defined in the procedural language reference manual – or natively defined. Native functions are those which are built into the Euclideus system, they are permanent and unmodifiable.

A function which is defined through the programming syntax is an entity that accepts a sequence of arguments, encloses a sequence of instructions, and returns another object. While the syntax for defining and invoking functions is completely detailed in the procedural reference manual, the object-oriented paradigm adds a subtlety to a function's scope characteristics depending on whether it is a *global* function (defined outside of any object or class) or a *member* functions (defined within an object or class).

The scope properties of global functions are exactly as described in the procedural reference manual: they have access to all objects defined outside of them but are unable to permanently change them. On the other hand, a member function is able to permanently change any object defined in its parent's scope. This will be clarified throughout the present manual.

Euclideus a set of native functions which are built into the system. These may be defined as global functions or as member functions, and each has its own set of properties and capabilities. Thus native functions are all described in this section.

1.2.1 The write() Function

The `write()` function is used to serialize and output an object to the filesystem. This function is a member of all Euclideus objects. `write()` accepts exactly one argument, a `string` describing the name of the output file, e.g. the command `a.write("filename")` will serialize and write the object `a` to a file called "filename". The return value of this function is `1` upon successful output or `0` if it was unable to write the object.

1.2.2 The read() Function

`read()` is a globally-defined function used to import an object from the filesystem which was previously serialized using the `write()` function. It accepts exactly a single argument, a `string` representing the name of the file that contains the serialized object, and returns the de-serialized object, or an error message if the object could not be de-serialized. The command `b=read("filename")` therefore reads an object stored in a file called "filename" and stores it into `b`.

1.2.3 The delete() Function

As its name implies, `delete()` is responsible for removing stored objects or object members from the environment. This function accepts an indefinite number of arguments, which represents the list of all items to be deleted. It is defined both as a global function and as a member function.

Used as a global function, `n=delete(a1,a2,a(x),...)` simply removes the items – objects, classes or functions – `a1,a2,a(x),...` from the environment, if they exist. If no arguments are given to `delete()`, it removes *all* stored quantities. The return value is the total number of items that were removed.

As a member function, `delete()` is used to remove members of an object[2]. Given some object `mut`, the command `n=mut.delete(a1,a2,a(x),...)` removes the members `a1,a2,a(x),...` from the object `mut` (if they exist) and stores the number of removed items in the variable `n`.

As a member function, note that `delete()` permanently modifies its parent object.

1.2.4 The mutate() Function

The `mutate()` function possesses the same characteristics as `delete()`, except that it only operates on objects[3]; instead of deleting them, it *mutates* them. Once again, objects which are instances of a class do not contain mutable members, and therefore their members cannot be mutated. Also, some native objects (e.g. those of `string` type) contain immutable members, which means that the `mutate()` function has no effect in such contexts. Generally, if an object is mutable, it can be transformed into a `mutant`-type object via the `mutate()` function. This entails a certain number of transformations which are described later in the present manual.

1.2.5 The print() Function

Any object is able to output a `string` representation of its structure – this includes all its class, object and function members (both public and private encapsulations are indicated) – as well as its type name. This is achieved using the object's `print()` function. Simply call `a.print()` (no arguments are given to this function) to return complete list of a's members in the form of a `string`-type object.

1.2.6 The introspect() Function

For some object types, such as `string`, `array` and `primitive` objects, the `print()` function returns a simplified representation of the object which may omit some of the object's specific characteristics, including its own functions. For example, the command `"Hello World!".print()` simply returns the `string` representation `"Hello World!"`, concealing all other properties, functions and structure of that `string` object. In order to reveal an object's complete set of properties, the `introspect()` function is provided. This function takes

[2]The members of an instance object cannot be mutated nor deleted, since these are defined by a class. In order to be able to delete members of an instance object, it must first be mutated, which is a procedure described below

[3]Functions and classes cannot be mutated.

no arguments and returns a `string` object which is a complete representation of an object's set of data and function members.

1.2.7 The type() Function

Objects are able to report their own type name – this is the name of their class if they are instance objects, `mutant` if they are mutant objects or the name of their native type (e.g. `string`, `array`, etc.) if they are native objects – as a `string`. `type()` does not accept any arguments, and returns an object of type `string`. For example, "test string"`.type()` would return "string", `3.type()` returns "primitive", and so on.

2 Native Objects

In Euclideus, a native object is one which is built into the system. It is not however an instance of any class, therefore its properties cannot be inherited, yet it still reports a type. Moreover a native object may introduce its own reserved syntax or keywords. To illustrate these ideas, Euclideus currently provides one such type of object: the `array` type.

2.1 The array Object

An array is a data structure that simply contains a set of elements (objects) juxtaposed one next to each other as shown in Fig. 2.1. Such an object is created using syntax similar to instantiation, the assignment

`a = array:100;` (2.1)

constructs an array object with `100` elements, each initialized to a primitive value `0`, and assigns the object to `a`. Although the syntax (2.1) appears to instantiate an object of a class called `array`, it is important to note here that Euclideus offers no such `array` class; `array` objects are classless, although they do have a type (the `array` type).[4]

The array's elements are accessed in constant O(1) time by using their *index*, which is just a number representing their position. As these elements are all public data members of the array object, they are accessible via indirection using the dot ("`.`") operator. As illustrated in Fig. 2.1 for a given array `a`, calls to

`a.0, a.1, a.2` (2.2)

yield the first, second and third elements, respectively. Any array element is thus obtained via indirection from the array object in such a way that the indirection evaluates to a number. For instance, a call such as `a.j` will yield an array element provided that `j` evaluates to a number that is between 0 and the largest array index (inclusively), the latter being equal to

[4]For instance objects, the object's type matches its class name. However, for general objects that are not instances, there is no class associated to the object's type.

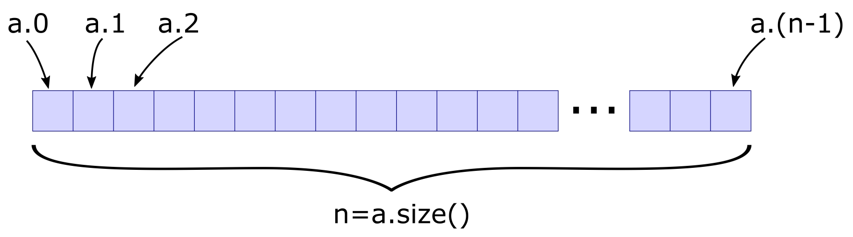


Figure 2.1: An array `a` contains a finite set of mutable elements which are objects. The array elements are in fact data members of the array object and therefore are accessed using the indirection ("`.`") operator using the element's index: `a.m` yields the element at index `m`, as shown in the figure above. Indices are numbered starting from `0`. The array's `size()` function returns the total number of elements in the array.

the array's size minus one.

An alternative way to create an array with an initial set of elements is provided by invoking a different array constructor. For example, the syntax for creating an array containing $N+1$ objects `x0, x1, ..., xN`, (in that order) is as follows:

```
a = array:x0, x1, x2, ..., xN;                    (2.3)
```

The constructor invoked in (2.3) will generate an array of $N+1$ elements, `a.0, a.1, ..., a.N`, and will assign to these elements the objects `x0, x1, ..., xN` in the corresponding order. If (i) only a single argument is provided to the array constructor, and (ii) the argument evaluates to a number, then the interpreter will invoke constructor (2.1) instead of (2.3), thereby interpreting the argument as the size of the array to be created.

As a further syntax simplification, `array`s may be more conveniently constructed using square brackets ("`[`" and "`]`") instead of the "`array:;`" constructor syntax. Thus, the expression `a=[x0, x1,x2, ...xN]` has the exact same effect as (2.3) above.

2.1.1 Getting the array Size

The array object also provides a public function that returns its size: simply call `a.size()` to obtain the size of an array `a`. This function may be used, for example, to retrieve elements located at the end of the array:

```
a.(a.size()-1), a.(a.size()-2), a.(a.size()-3)          (2.4)
```

will retrieve the last three array elements.

2.1.2 Enlarging an array

If an array must be enlarged from an original size `n`, `a.enlarge(N)` will increase the size of an array `a` to a larger size `N`, filling positions `n` to `N-1` with `0`s (primitive-type objects). If `N` is not greater than the original array size `n`, or if it is simply undefined, no resizing action is performed. In addition to modifying the array, this function also returns a reference to the newly-resized array. Consider the following commands as a typical usage scenario:

```
  a = array:10;          (2.5)
a.0 = -33/pi,            (2.6)
a.9 = i^i,               (2.7)
  b = a.enlarge(100)     (2.8)
```

Here, the first command (2.5) creates an array `a` containing ten elements, each carrying the value `0`, after which (2.6) and (2.7) assign values of `-33/pi` and `i^i` to the first and last array elements, respectively. On line (2.8), the enlarging function performs two functions: (i) it increases the size of the array `a` to one hundred elements, filling it with `0`s from position 10 to 99; (ii) it then returns a reference to the new array (of size 100) which is copied into a new array object `b`. Immediately after the execution of (2.8), `a` and `b` contain identical sets of elements, while being separate objects. Therefore, any operation(s) performed on one of the arrays will not affect the other.

As mentioned, when an array object is initialized its elements all carry the value `0`, meaning that the elements are all initially of primitive type. However unlike instance objects, each member of an array – in other words each one of its elements – is individually mutable such that any array element may be assigned to a different type, allowing its type to change.

2.1.3 Appending Items to an array

If additional elements are to be added to the array's current set, the `array` object provides an `append()` function. Unlike the previously discussed functions, the `append()` is able to accept a variable number of arguments, its behaviour depending on the number and nature of the arguments. If a single argument `obj` is provided, the command `append(obj)` will either

1. (i) append the single object contained at the end of the array, thereby incrementing the array size by one unit, or
2. (ii) if `obj` is itself an array, it will append each of `obj`'s elements to the array, thus increasing the array size by `obj.size()` elements

On the other hand, when more than a single argument is given to `append()`, each argument will be added as an additional array element (regardless of whether it is an array or not). Just like the `enlarge()` function, `append()` also returns a reference to the newly modified array, which may be copied into another object. As an example, consider the three following cases:

```
a = array:x,y,z;                    (2.9)
b = a.append(array:100;)            (2.10)
c = b.append(u,v,w,array:p,q,r;)    (2.11)
```

On line (2.9), an array a is created with three elements, x, y and z. This is followed by a call to a's append() function, which is given a single argument, array:100;. This argument is an array constructor which builds an array of one hundred elements and inputs that array object as a single argument to the append function. As a result, the function interprets its input as a single argument of array type, and therefore adds each element of the input array to a. As a result, a's size is increased to 103 elements. Following the enlargement of a, the assignment instruction copies the entire (enlarged) array to a new object b. Thus after executing (2.10), b and a are identical arrays of 103 elements – their first three elements are x, y and z, followed by zeroes.

On line (2.11), the append() function is invoked from object b, and is this time given multiple arguments, including an array argument. As stated, when multiple arguments are provided to the append() function, each becomes appended as an additional array element without being further broken down. Hence, the append() function on (2.11) adds four elements to b, and copies the entire array to a new object c. As a result og (2.9-2.11), a contains 103 elements while b and c are identical arrays of 107 elements.
The following example aims to synthesize all of the above concepts while specifically elucidating array element mutability by creating a matrix – which is an array of array objects:

```
matrix = array:100;                          (2.12)
for:j=0, j<matrix.size(), j=j+1,             (2.13)
    matrix.j=array:500;                      (2.14)
;                                            (2.15)
```

In the above, the first statement (2.12) assigns an array of 100 elements to an object called matrix. In other words, after line (2.12) matrix is an ordinary one-dimensional table of 0s as exemplified in Fig. 2.1 (its elements are all of primitive type). The purpose of the loop (2.13-2.15) is to mutate matrix's elements so that each of them individually becomes an array in its own right. This action is performed simply by reassigning an array (here, of size 500) to each of matrix's elements. Thus, after (2.15) matrix is a two-dimensional table of 100 × 500 elements – or in the language of objects, matrix is an object containing 100 members, and each member of matrix is itself an object (of type array) containing 500 primitive-type members. Using the provided indirection syntax, the $(m+1)$th primitive element of the $(n+1)$th array element of matrix is retrieved simply by calling matrix.n.m, which contains two successive indirections from matrix: a first one (".n") yields the $(n+1)$th array with a subsequently indirection (".m") to obtain its $(m+1)$th element.

This scheme may be further generalized to generate cubes and hypercubes of data by simply going over each primitive element in matrix and assigning an array to it, analogously to the prescription given in (2.12-2.15). More generally, since array elements may themselves be arrays of variable sizes, data sets of arbitrary topology can be generated in this manner.

2.2 The matrix Object

While `array` objects can in principle be employed as matrices, they are not optimized for matrix operations. As such, Euclideus supplies a `matrix` type which is more suitable for such computations involving multidimensional numerical arrays – scalars, vectors, matrices, and higher-rank numerical objects.

2.2.1 matrix Syntax

A `matrix` object can be initialized via the usual constructor syntax; e.g. `v=matrix:100;` creates a one-dimensional matrix (a vector) of `100` complex-valued elements, each initialized to `0`, `m=matrix:10,20;` creates a complex-valued rank-2 object (a matrix) containing `10` rows and `20` columns with all elements initialized to `0`, and so on.

As with `arrays`, a `matrix` may also be directly initialized using the square bracket ("`[`" and "`]`") syntax. Consider the following example:

```
v1 = [1,2-i,3/i,sqrt(-2-i)]        (2.16)
v2 = [-1.3,2+i,0,exp(i*3)]         (2.17)
v3 = [-1.3,2+i]                    (2.18)
m1 = [v1,v2]                       (2.19)
m2 = [v3,[2,6]]                    (2.20)
```

On (2.16-2.18), three rank-1 `matrix` objects are created and assigned to `v1`, `v2`, and `v3`. Then on (2.19), a rank-2 `matrix` is formed by assigning `v1` and `v2` as the first and second row, respectively, of the rank-2 object, which is then assigned to `m1`. On (2.20), a 2x2 `matrix` is created with `v3` as its first row and `[2,6]` as its second row.

As a first remark, note that all elements placed between the square brackets necessarily evaluate to numbers or to `matrix` objects of identical rank and size; numbers being rank-0 objects, `v1`, `v2` and `v3` being rank-1 objects and `m1` being a rank-2 object. If these conditions are not fulfilled, an `array` is created instead of a `matrix`. Thus, expressions such as `[m,v1]` or `[v1,v3]` would yield `array` objects rather than `matrix` objects.

The elements of a `matrix` may be accessed via indirection using the exact same syntax as with `array` objects. Given the assignments (2.16-2.20), `v1.3` yields the number `0.34356-1.4553*i`, `m2.1.0` yields 2, `m1.1.(m2.1.0)` yields `-3*i` etc. Note that `m2.1` returns the `matrix [2,6]` – the entire second row of `m2`. In general, the number of indirections from a `matrix` defines the rank of the returned object.

Assigning new values to `matrix` elements is accomplished with the usual syntax involving the assignment ("=") operator. For example, the assignment `m2.0=[-4,8+i]` replaces the first row of `m2` with a set of new elements, transforming `m2` to the form

$$\begin{pmatrix} -4 & 8+i \\ 2 & 6 \end{pmatrix} \tag{2.21}$$

with the first row replaced with the RHS of the assignment statement. Note that such an assignment only succeeds if it preserves the `matrix`'s integrity (its rank and its dimensions) – the new element must have the same rank and dimensions as the other `matrix` elements, otherwise the assignment fails. Consider `m2.1=[3,-4,5,6]`, `m2.1=8` and `m2.0=[[-4,8+i]]` as examples of unsuccessful assignments.

In addition to the above syntax, `matrix` objects come equipped with a set of member functions supporting various operations pertaining to matrices. The following sections detail the specific usage of each `matrix` function. While the examples listed in this section mainly use rank-1 and rank-2 `matrix` objects for the sake of clarity, please note that all functions and `matrix` operations described in this section support objects of arbitrary rank.

2.2.2 MATRIX ADDITION AND SUBTRACTION

Adding or subtracting two `matrix` objects is accomplished, respectively, via the `add()` and `sub()` functions which are both members of the `matrix` object. Each function accepts a single argument: the `matrix` to be added to or subtracted from the parent object, and returns a new matrix which is the sum or the difference between the two objects. For example, `[1,2,4].add([4,5,6*i])` returns a new rank-1 object: $(5,\ 7,\ 4-6\mathrm{i})$.

Assuming both matrices have the same rank and dimensions, `matrix` addition and subtraction follow the usual definition, e.g.

$$\texttt{[[1,2],[3,4]].sub([[i,-3],[6,8*i]])} \Leftrightarrow \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} - \begin{pmatrix} i & -3 \\ 6 & 8\mathrm{i} \end{pmatrix}, \tag{2.22}$$

where corresponding elements are simply added or subtracted.

Euclideus also allows for adding or subtracting two `matrix` objects of different ranks. In this case, the lower-rank object is taken as an element of an augmented version whose dimensions are the same as those of the other object. To see this more clearly, consider a special case of a scalar (rank-0) object subtracted from a rank-2 `matrix`:

$$\texttt{[[1,2,3],[4,5,6]].add(5*i)} \Leftrightarrow \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} + \begin{pmatrix} 5\mathrm{i} & 5\mathrm{i} & 5\mathrm{i} \\ 5\mathrm{i} & 5\mathrm{i} & 5\mathrm{i} \end{pmatrix} \tag{2.23}$$

On (2.23), the parent object is a 2×3 `matrix` while the argument given to `add()` is of rank `0` (a number). As a result, Euclideus interprets the lower-rank object as a 2×3 `matrix` whose elements are all equal to the provided scalar argument, `5*i`; and returns the sum of these two matrices.

Now consider the following example of a vector subtracted from a rank-2 `matrix`:

$$\texttt{[[1,2,3],[4,5,6]].sub([5*i,3,-2])} \Leftrightarrow \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} - \begin{pmatrix} 5i & 3 & -2 \\ 5i & 3 & -2 \end{pmatrix} \tag{2.24}$$

2.2.3 matrix Multiplication

Ordinary `matrix` multiplication is provided through the `mul()` function, a function member of the `matrix` object. The `mul()` function takes a single argument – a `matrix` multiplying the parent object *to the right* – and returns a new `matrix` object representing the product of the parent `matrix` and the one passed as an argument.

The manner in which an operation `A.mul(B)` is interpreted depends upon the dimensions of `A` and `B`. If the number of columns in the parent object `A` matches the number of rows in the argument `B` passed to `mul()`, a standard `matrix` product `AB` is produced; if one of the objects is a scalar `matrix`[5], the operation becomes a multiplication of a `matrix` by a scalar; if the two objects are of rank 1 and of equal size, the operation is understood as a dot product. To illustrate these cases, consider the following examples:

$$\texttt{[[1,2],[3,4],[5,6]].mul([[6,5,4],[3,2,1]])} \Leftrightarrow \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \begin{pmatrix} 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix} \tag{2.25}$$

$$\texttt{[[1],[2],[3]].mul([[4,5,6]])} \Leftrightarrow \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \begin{pmatrix} 4 & 5 & 6 \end{pmatrix} \tag{2.26}$$

$$\texttt{[1,2,3].mul([4,5,6])} \Leftrightarrow \left(\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \bullet \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \right) \tag{2.27}$$

$$\texttt{[[1,2,3],[4,5,6],[7,8,9]].mul([[i]])} \Leftrightarrow \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \begin{pmatrix} i & 0 & 0 \\ 0 & i & 0 \\ 0 & 0 & i \end{pmatrix} \tag{2.28}$$

The first two examples correspond to ordinary `matrix` multiplication. On (2.25), a 3×2 `matrix` (three rows and two columns) is multiplied by a 2×3 `matrix`, which is naturally interpreted as an ordinary `matrix` product. Likewise, on (2.26), a 3×1 column `matrix` is multiplied to the right by a 1×3 row `matrix`, again resulting in a 3×3 square `matrix`.

Now, on (2.27), the `mul()` function is called on two rank-1 objects of equal size. In this case, the ambiguity is lifted by treating it as a vector dot product between the two objects, which will result in a scalar `matrix` of size one containing the value $1 \times 4 + 2 \times 4 + 3 \times 5 = 32$.

[5] In Euclideus, any `matrix` of size 1 in each of its dimensions is interpreted as a scalar.

On (2.28), a 1×1 `matrix` is passed as an argument to `mul()`, which is called from a 3×3 parent `matrix`. In this case, the 1×1 object is understood as a scalar `matrix`, i.e. a `matrix` which is proportional to the identity `matrix`. The same happens if a scalar (rank-0) object is passed as an argument to `mul()`.

In order to further simplify the syntax, `mul()` is able to operate on objects of different ranks. For instance, when a rank-2 `matrix` is multiplied by a vector, the vector is interpreted as a row `matrix` if it multiples to the left and a column `matrix` if it multiplies to the right. The following example illustrates this more clearly:

$$\texttt{[[1,2],[3,4]].mul([5,6])} \Leftrightarrow \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}\begin{pmatrix} 5 \\ 6 \end{pmatrix} \tag{2.29}$$

$$\texttt{[5,6].mul([[1,2],[3,4]])} \Leftrightarrow \begin{pmatrix} 5 & 6 \end{pmatrix}\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \tag{2.30}$$

On (2.29), the vector is naturally treated as a column `matrix` when multiplying on the right, and on (2.30) as a row `matrix` when multiplying on the left.

2.2.4 Direct Product and Sum of matrix Objects

The `matrix` object supplies the functions `diradd()` and `dirmul()` to evaluate the direct sum ($\oplus$) and direct product ($\otimes$) between `matrix` objects. While the properties of these operators may be found in various mathematics textbooks, their action is illustrated in the following examples:

$$\texttt{[[1,2],[3,4]].dirmul([[1,2,3],[4,5,6],[7,8,9]])} \Leftrightarrow \begin{pmatrix} 1\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} & 2\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \\ 3\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} & 4\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \end{pmatrix} \tag{2.31}$$

$$\texttt{[[1,2],[3,4]].diradd([[1,2,3],[4,5,6],[7,8,9]])} \Leftrightarrow \begin{pmatrix} 1 & 2 & 0 & 0 & 0 \\ 3 & 4 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 3 \\ 0 & 0 & 4 & 5 & 6 \\ 0 & 0 & 7 & 8 & 9 \end{pmatrix} \tag{2.32}$$

2.2.5 Other Functions for Performing Standard Evaluations on a matrix

This section briefly describes an additional set of functions, each taking no arguments, provided for common computations on a `matrix` object `M`.

(i) `M.conj()` conjugates all the elements of `M` and returns a reference to the parent object. `M` remains modified after completion of this operation.

(ii) `M.transpose()` transposes `M`, changing its rows to columns and columns to rows. This operation is performed recursively on the elements if they are themselves of non-zero rank. `M` remains modified (as its transpose) after completion of this operation.

(iii) `M.invert()` calculates the inverse $\mathtt{M}^{-1}$ and returns a reference to the inverted object. `M` remains modified (as its inverse) after completion of this operation.

(iv) `M.det()` returns the determinant of `M`, and `M.trace()` returns the trace of `M`.

(v) `M.eigvecs()` returns a `matrix` whose rows are the eigenvectors of `M`, normalized to unity. The row-order of the eigenvectors corresponds to the order of the eigenvalues returned by `eigvals()`.

(vi) `M.eigvals()` returns a vector containing the eigenvalues of `M`, the order of the eigenvalues corresponds to the row-order of the eigenvector `matrix` returned by `eigvecs()`.

(vii) `M.rank()` and `M.size()` return, respectively, the rank and the size of `M`.

(viii) `M.dims()` returns a vector, of size `M.rank()`, containing the sizes of each dimension of `M`, beginning from the outermost layer to the innermost layer. For example, if `M=[m,m,m,m]`, where each `m` is a 2 × 3 (rank-2) `matrix` object, then `M.dims()` returns a vector `[4,2,3]`.

2.2.6 Additional Utility Functions of the matrix Object

This section lists a set of less mathematical functions designed to aid in managing and modifying `matrix` objects.

(i) `M.append(m1,m2,m3)` is a function taking a variable number of arguments, each of identical dimension which coincides with the dimension of `M`'s elements. The function appends the arguments at the end of `M` in the given order and returns a reference to `M`, which remains modified after completion of this operation. For example, if `M` is a vector (rank-1) the added elements are scalars; if `M` is a rank-2 `matrix` object, `m1`, `m2` and `m3` are added as rows to `M`.

(ii) `M.appendcol(c1,c2,c3)` is a function also taking a variable number of arguments. Here, the arguments are understood as new columns to be appended to `M`, their rank is one less than that of `M`, and their size is necessarily equal to the size of `M`, i.e. its number of rows. For example `[[1,2],[3,4],[5,6]].appendcol([2,4,6],[1,3,7])` appends two columns to a 3 × 2 `matrix`, resulting in a 3 × 4 `matrix`.

(iii) `M.delete(i1,i2,i3)` is a function of a variable number of arguments which are integers representing the indices of elements to be deleted from `M`. The function returns a reference to the modified `matrix` `M`. For example, `[[1,2],[3,4],[5,6]].delete(1,2)` removes the second and last rows from the parent object, resulting in a 1 × 2 `matrix`.

(iv) `M.deletecol(i1,i2,i3)`, also accepting a variable number of arguments, removes the columns of `M` at indices `i1`, `i2` and `i3`. For example, `[[1,2,3][4,5,6]].deletecol(0,2)` removes the first and last column of a 2×3 `matrix`, resulting in a 2×1 column `matrix`. The parent object remains modified upon completion.

(v) `M.array()` creates a new `array` object from the `matrix` `M`. The elements of the new `array` are copies of the elements of `M` but are mutable, as per the properties of `array` objects.

(vi) `M.vectorize()` takes no arguments and produces a new `matrix` which contains all the elements of `M` placed linearly inside an object of reduced rank. Thus, for instance, `[[1,2],[3,4]]`.vectorize() creates and returns the vector `[1,2,3,4]` from a rank-2 2×2 `matrix`.

(vii) `M.matricize()` takes no arguments and produces a reduced-rank `matrix` from the parent object. If `M` is a rank-$2n$ object, with $n > 1$, `M.matricize()` creates and returns a rank-$(2n-2)$ `matrix` by interpreting `M` as a "matrix of matrices". As an example, consider `M=[[m1,m2],[m3,m4]]`, a rank-4 object containing the rank-2 objects `m1`, `m2`, `m3` and `m4`, all dimensions 3×2. `M.matricize()` then returns a 6×4 rank-2 object by simply juxtaposing the columns and rows of the four sub-matrices in a new larger rank-2 `matrix` according to the row and column order they appear in `M`.

(viii) `M.writematrix(path,separator)` complements the standard `write()` function by outputting `M` as a human-readable file if its rank is below 3, i.e. if it is a scalar, a vector or a rank-2 `matrix`. The function takes either one or an optional second argument, both `strings`. The first argument represents the relative path of the output file and the second argument is the separator to use either (A) between columns for a rank-2 object (the default separator being a space) or (B) between elements for a rank-1 object, the default separator being a new line. For instance, `[[1,2],[3,4]].writematrix("m1.txt",";")` outputs the parent 2×2 `matrix` to a file called "m1.txt" with columns separated by semicolons, while the columns are separate by a space if the second argument is omitted.

(ix) `readmatrix(path,separator)` reads in a rank-2 `matrix` from the file system and returns the corresponding `matrix` object. The function arguments are identical to those of `writematrix()` described in item (viii) above, except that the second argument is always understood as the column-separator (by default a space if the second argument is omitted) and rows are always separated by a new line.

(x) `readvector(path,separator)` reads in a rank-1 `matrix` (a vector) from the file system and returns the corresponding `matrix` object. The function arguments are identical to those of `writematrix()` described in item (viii) above, except that the second argument is always understood as the element-separator (by default a new line if the second argument is omitted).

2.3 The `string` Object

In order to facilitate the processing of ordinary text, Euclideus supplies the `string` object which includes a number of utility functions. In most programming languages, a standard string is usually understood as a one-dimensional array of characters (bytes) stringed together into an array, usually with a null terminating character which serves to delimit the string. Euclideus recognizes the quotation mark (") as a delimiting bracket for `string` literals, i.e. the sequence of characters enclosed by a pair of quotation marks is to be taken literally – interpreted character by character – without any further syntactical processing from the interpreter. As a basic example, consider the following instruction,

`s1 = `"This is a string called \"test\"." (2.33)

In (2.33), a `string`-type object is constructed on the RHS and stored into a variable called `s`. Note the particular pair of \" symbols around the word “test”. These are called escape sequences and are used to allow for the proper representation of special characters – characters that are otherwise reserved for special uses – inside a `string`. As in many other languages, Euclideus appropriates the backslash (\) character as a means to escape such special characters. Any special character that follows the backslash will be interpreted literally in the `string`. In the above example, this means that the “\"” sequences will each be interpreted as single quotation mark characters around the word “text”. Since the backslash character is reserved as an escape character, this means that it must also be escaped inside a `string`, e.g.

`s_esc = `"The escaped backslash “\\” sequence is interpreted as a single backslash." (2.34)

As a result, the `string` stored in `s1` will have a character count of 31 while the one shown in (2.34) will contain a total of 72 characters, the escaped sequences in each case being understood as single characters. Moreover, the delimiting (non-escaped) quotes are not interpreted as part of the `string`, as they are understood within Euclideus' formal syntax. The total character count of a `string` is provided by the `charcount()` function, i.e. `s_esc.charcount()` would return the value 72 while `s1.charcount()` would return 31.

The examples (2.33) and (2.34) produce `string` objects which may be understood as arrays of characters, or in other words arrays of `strings` of length one. As such, the elements of a `string` may be accessed in a manner similar to that of arrays, using an indirection to a number representing the index of the element in the array. For example, `s1.0` will yield "T" (a `string` of length one) and `s1.24` will yield "\"", again a `string` of length one (keeping in mind that an escaped character counts as one character).

2.3.1 Appending Text to a string

One of the capabilities bundled with the native `string` type is the ability to append additional `strings` at the end of the object. This is achieved using the `string`'s own `append()` function; its behaviour is similar to the `array`'s `append()` function in that it can accept an indefinite number of arguments, appends the `strings` contained in the argument list to its parent `string` object, and returns a reference to the modified `string`. Consider the following example, which recalls `s1` as defined in (2.33):

`s2 = s1` (2.35)

`s2.append(" The '","2/sqrt(-1)","' in this appendix"," is imaginary.")` (2.36)

`s3 = s1.append(" The '",2/sqrt(-1),"' in this appendix"," is imaginary.")` (2.37)

Here, in each case the `append()` function is given four arguments. On (2.36) the arguments are all `strings` while on (2.37), the second argument (`2/sqrt(-1)`) does not possess quotation marks around it which means that it evaluates to a `primitive` object. In general, whenever a non-`string` object is passed to one of the `string` functions, it is first evaluated to a `string` type using the object's built-in `print()` function (this function outputs a `string`-representation of the object). In the case of a `primitive` object, the interpreter first mathematically evaluates the expression contained in the object, and then calls its `print()` function which provides a `string`-representation of the expression.

Since the `append()` function acts on its own parent object, (2.36) and (2.37) modify `s2` and `s1`, respectively, and on (2.37) the reference to `s1` returned by the `append()` function is copied into `s3`. Thus upon executing (2.35-2.37), `s2` and `s3` (the latter which is a copy of `s1`) will contain the following text:

`s2` ←"This is a string called \"test\". The '2/sqrt(-1)' in this appendix is text." (2.38)

`s3` ←"This is a string called \"test\". The '-2*i' in this appendix is imaginary." (2.39)

2.3.2 A STRING AS AN ARRAY OF STRINGS

In the examples considered so far, text has been represented simply as an array of characters, interpreted literally by enclosing characters between the quotation marks. The quotation marks may be understood as an object constructor, building a `string` object as an array of single-character `strings`. Similarly, the `string`'s `append()` function decomposes each input `string` into single characters and adds them in sequence to the parent `string`.

In order to achieve greater functionality in text processing, Euclideus generalizes the in-memory representation of text by defining a native `string` object type recursively as follows:

A `string` object is (i) an array of `string` objects unless (ii) it is a null `string`.

In the above definition, (i) is the recursive part and (ii) is the base case, for which a null `string` is one containing no characters[6]. The key contribution of the above generalization is that it allows for text to be represented as a multidimensional array of `string` objects instead of a one-dimensional table of characters. In object-oriented terminology, a `string` is an object

[6]In most languages, such a `string` would be represented as an array of one character, being the null character. A character count of such a `string` normally yields a value of zero, as the terminating null character is not considered in the `string`'s character count.

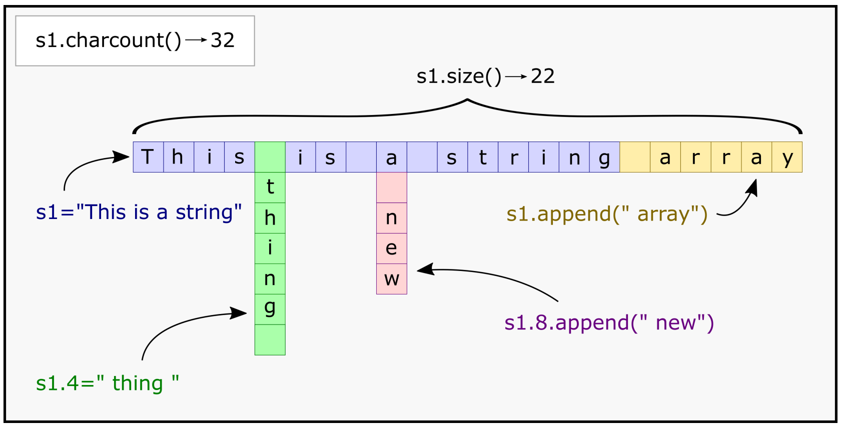


Figure 2.2: A general `string` object contains elements (object members) which are `string` objects themselves. This figure illustrates a few of the operations that may be performed on a `string`.

whose members are all of `string`-type and (unlike the `array`) are all *immutable* – the members of a `string` cannot be of a type other than `string`. To demonstrate these properties, let us consider the following example (illustrated in Fig.2.2):

$$\texttt{s1} = \text{"This is a string"} \tag{2.40}$$

$$\texttt{s1.append(}\text{" array"}\texttt{)} \tag{2.41}$$

$$\texttt{s1.8.append(}\text{" new"}\texttt{)} \tag{2.42}$$

$$\texttt{s1.4} = \text{" thing "} \tag{2.43}$$

Starting from (2.40), `s1` is initialized as a simple array of characters enclosed within the quotation brackets. Next, a `string` is appended to `s1` using the `append()` function, which simply decomposes the argument (" array") into individual characters and tacks them onto the end of `s1`. Now keeping in mind that all elements of `s1` are `string` objects themselves, on (2.42) the `append()` function is invoked from the ninth element of `s1`, appending " new" at the end of `s1.8`. This action amounts to inserting a `string` between the ninth and tenth elements of `s1`. Finally, on (2.43) the fifth element of `s1` (a space character) is replaced with " thing " by reassigning it to another `string` object. Even though the members of a `string` are immutable, this is nevertheless possible because no actual mutation occurred: the new value of `s1.4` is still of `string` type despite being of different dimensions.
The text contained in `s1` is obtained by recursively traversing the multidimensional `string` in the order of its elements. Thus, after executing (2.40-2.43), `s1`'s text will read "This thing is a new string array", with a total `s1.charcount()` of 32 characters. This example also serves to emphasize the difference between the `charcount()` function and the `size()` function. While the former simply provides the total number of characters in the string regardless its in-memory topology (which is essentially the amount of memory occupied by the string),

the `size()` function operates exactly as it does for `array` objects: it yields the size of the `array`, ignoring any substructure in its elements. In the above example, it can be seen that `s1.size()` will yield a value of 22.

In the same fashion as in the example above, additional strings may also be appended or assigned to the fifth and ninth elements (" thing " and " new", respectively). For instance, `s1.8.4.append("er")` will append two extra characters at the end of the last element of `s1`'s ninth element which would make its topology grow into the third dimension; `s1` would then read "This thing is a newer string array".

To leverage this multidimensional structure, Euclideus allows direct access to the `string:;` constructor syntax, and provides a few more string functions: `tochararray()`, `appendw()`, `split()` and `replace()`.

2.3.3 Appending Words to a string

Like the basic `append()` function, `appendw()` is designed to add content at the end of a `string`. It also accepts an indefinite number of arguments, and processes the arguments in the same manner as `append()`. However, instead of appending the arguments' characters individually one-by-one at the end of the parent `string`, it appends each argument as a separate *word*, thereby immediately growing the string in another dimension. To better see how this works, consider the following example:

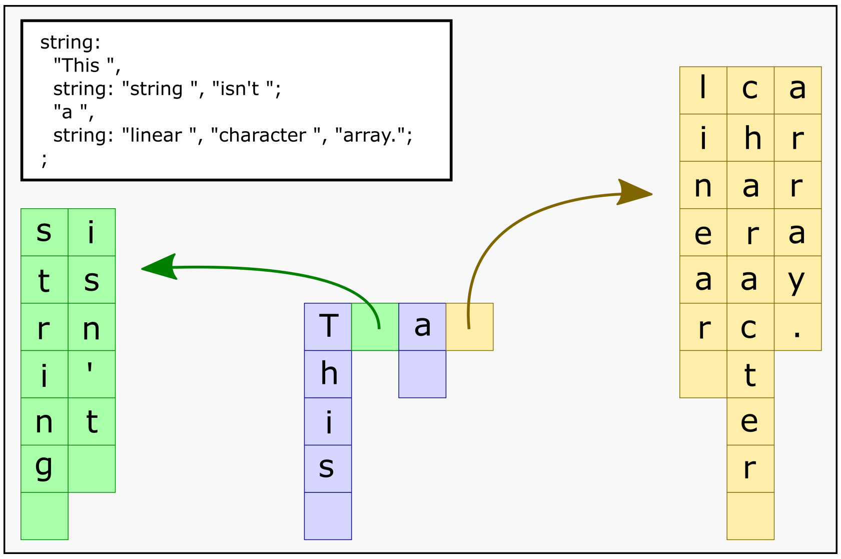
```
w = "These "                                                    (2.44)
w.appendw("words ","are ","individually ","appended.")          (2.45)
```

On (2.44), an ordinary string is initialized as an array of six characters. This is followed on (2.45) by a call to `w`'s `appendw()` which adds four elements – each as character arrays – to `w`. In other words (no pun intended), the last four elements of `w` are arrays of size `6`, `4`, `13` and `9`, respectively. As a result of (2.45), `w.size()` returns `10` while `w.charcount()` returns 38; the text contained in `w` reads "These words are individually appended."

2.3.4 The string Object Constructor

Creating a `string` using the quotation marks produces an array of characters (an array of `string` objects of unit length). As shown in Fig. 2.2, Euclideus allows `string` objects to be more complex structures than simple unidimensional arrays. In order to leverage this capability, strings may be be created by adopting the standard constructor syntax, using the `string:;` object constructor. This constructor accepts an indefinite number of parameters which are used to initialize the `string`.

As the elements of `string` are all necessarily `string` objects themselves, all non-`string` parameters given to the constructor are converted to `strings` as they are inserted into the new



```
string:
  "This ",
  string: "string ", "isn't ";
  "a ",
  string: "linear ", "character ", "array.";
;
```

Figure 2.3: Using the `string:;` constructor directly allows for the creation of multidimensional `string` objects from the outset. Each argument passed to the constructor is interpreted as an individual `string` object.

object, while all `string`-type parameters are inserted into the object *in their original topology*. This allows the `string` object constructor to initialize a `string` with an arbitrary topology. To illustrate this property, consider the following example:

```
sc = string:                                              (2.46)
       "This ",                                           (2.47)
       string:"string ","isn't ";                         (2.48)
       "a ",                                              (2.49)
       string:"linear ","character ","array.";            (2.50)
     ;                                                    (2.51)
```

Here, the `string:;` constructor is directly invoked to initialize a `string` as non-linear in-memory stricture. The structure, as displayed in Fig. 2.3, is an array of four elements which are initialized through the constructor's input arguments (2.47-2.50). Lines (2.47) and (2.49) initialize the first and third elements as simple character arrays, while (2.48) and (2.50) call the `string:;` constructor to initialize the second and fourth elements as arrays of character arrays themselves.

Each element and sub-element of `sc` may be accessed using the usual indirection syntax for arrays. For instance, `sc.0` will yield "This "; `sc.1` will retrieve the second element of `sc`, which is an array of character arrays; `sc.1.1` will yield "isn't " and `sc.1.1.(sc.1.1.size()-1)` will return a space (the last element in the array "isn't " is a space character).

2.3.5 Comparing a string's Lexicographic Ordering to Another

Although they are not numeric, `string` objects may nevertheless be compared in terms of their lexicographical (or dictionary) order. This is accomplished using the `string` object's `compare()`, which takes exactly one argument: a second `string` against which it is to be lexicographically compared. The `compare()` function works exactly as it does in the C language: it returns a negative value, zero, or a positive value (not a string but rather a `primitive` object) depending on the lexical ordering of the parent `string` and the argument given to `compare()`. In the example below, all boolean statements are true (they all evaluate to `1`):

```
"dog".compare("rat")<0,        (2.52)
"dog".compare("dog")==0,       (2.53)
"dog".compare("cat")>0,        (2.54)
```

In (2.54-2.52) parent object ("dog" in each case) is *compared against* the argument given to `compare()`. If it (i) falls before (after) the argument, in a dictionary sense, then `compare()` returns a negative (positive) value, while if the `strings` are equivalent then `compare()` returns `0`.

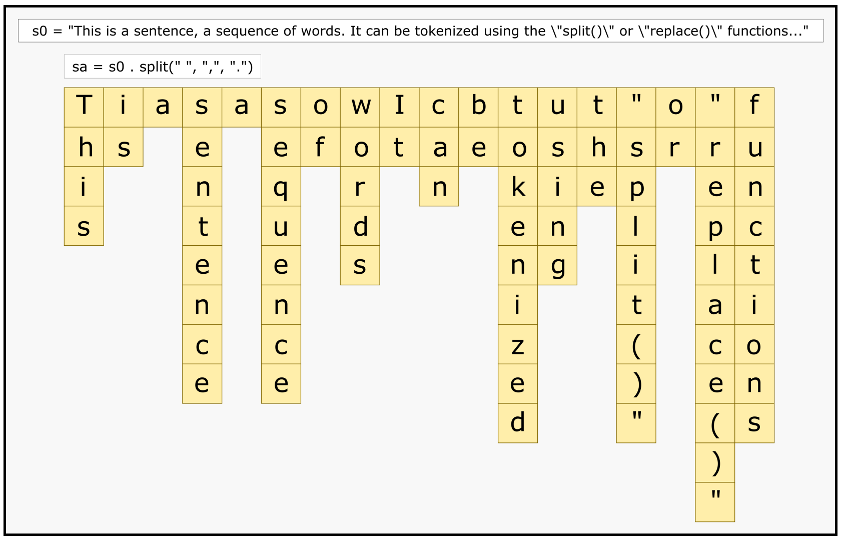
It is important to note here that `compare()` does not take into account the in-memory topology of either `string` object – they are compared to each other as simple character arrays. Thus, two `strings` which compare to zero may not contain the same topology.

2.3.6 Collapsing a string Object into a Linear Array of Characters

Since a Euclideus `string` object generally possesses an arbitrary topology, it may be useful to transform it into a simple linear array of characters. This is accomplished with the help of the `characterize()` function, which is a member of `string` objects. `characterize()` does not modify its parent `string`, instead it returns a new `string` object which is a version of its parent transformed into a simple array of characters. Using the example above, `sc2 = sc.characterize()` would yield a new `string` object `sc2` containing 43 elements, each one being a character (i.e. a string of unit length). Thus, `sc2.charcount()` and `sc2.size()` would both return 43 in this case.

2.3.7 Separating a String With Respect to a Set of Delimiters

It is often useful to separate text with respect to a certain set of characters or words which are used to demarcate the boundaries between sub-regions of text. To this end, Euclideus provides a `split()` function as a member function of `string` objects. The `split()` function accepts an indefinite number of arguments, each interpreted as `strings` and understood as delimiters used for splitting the parent `string`. Calling `split()` on a `string` therefore



```
s0 = "This is a sentence, a sequence of words. It can be tokenized using the \"split()\" or \"replace()\" functions..."
sa = s0 . split(" ", ",", ".")
```

Figure 2.4: A `string` `s0` is segmented into words (tokens) with respect to the space, comma and dot delimiters; these delimiters do not appear in the tokenized `string`.

separates it with respect to these delimiters. The following example is a typical usage scenario of the `split()` function:

$$
\begin{aligned}
\texttt{s0} &= \text{"This is a sentence, a sequence of words. It can be tokenized into words using} \tag{2.55}\\
&\ \text{the \textbackslash"split()\textbackslash" or \textbackslash"replace()\textbackslash" functions...",} \tag{2.56}\\
\texttt{sa} &= \texttt{s0.split(" ", ",", ".")} \tag{2.57}
\end{aligned}
$$

In this example, a `string` `s0` is split into individual words according to spaces, commas and periods, and the result is stored in `sa`. After processing (2.55-2.57), `s0` remains a simple character array as originally defined (2.55) using the quotation marks and `sa` is a `string` with a topology as illustrated in Fig. 2.4.

The delimiters passed as arguments to the `split()` function are not transferred to the new `string`, while all regions of text content lying between these delimiter arguments are inserted, in order, into the new `string` as character arrays. Thus the `split()` function performs two roles: (i) it transforms a `string`'s topology into a form that is prescribed by a set of delimiters, (ii) it deletes all occurrences of a set of substrings (the delimiters) from within a `string` object. Since all elements of a `string` object are `string`s themselves, the `split()` function may be carried out recursively on the elements.
As a final note, the `split()` function always yields a `string` having a topology as given in Fig. 2.4 – i.e. a sequence of character arrays – irrespective of the original `string`'s topology.

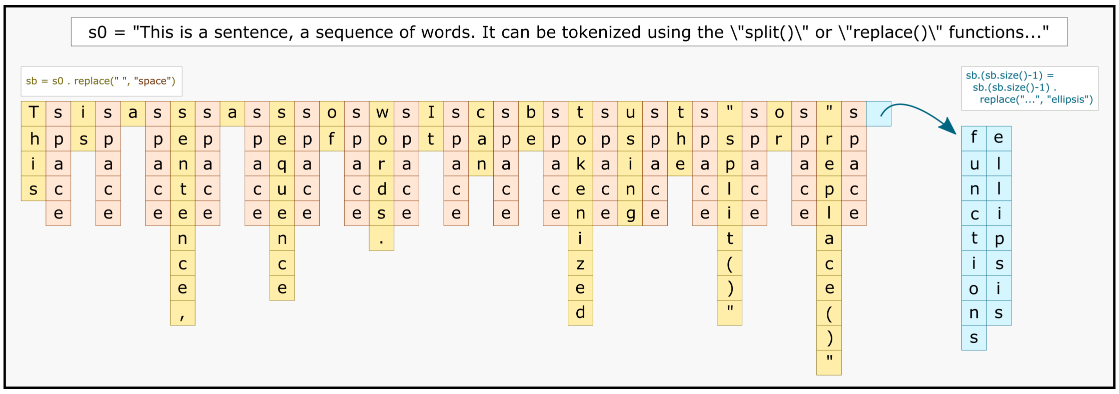

s0 = "This is a sentence, a sequence of words. It can be tokenized using the \"split()\" or \"replace()\" functions..."
sb = s0 . replace(" ", "space")
sb.(sb.size()-1) = sb.(sb.size()-1) . replace("...", "ellipsis")

Figure 2.5: The `string` `sb` is obtained from `s0` by performing two substring replacement operations: (i) a first operation where the space character is substituted with the literal "space" `string`, this splits it with respect to the space `string` which serves as a delimiter; (ii) the last element of the resulting `string`, `sb.(sb.size()-1)` is then changed by substituting the "..." sequence with the `string` literal "ellipsis".

2.3.8 Replacing Occurrences of a Substring

Euclideus `string` objects are equipped with a `replace()` function that allows for the replacement of a substring with a new one. This function accepts exactly two arguments: (i) the substring to be replaced and (ii) the new `string` that replaces it. The following example illustrates its usage:

$$\begin{aligned}
\texttt{s0} &= \texttt{"}\text{This is a sentence, a sequence of words.} && (2.58)\\
&\quad\ \text{It can be tokenized into words using} && (2.59)\\
&\quad\ \text{the \textbackslash"split()\textbackslash" or \textbackslash"replace()\textbackslash" functions..."}, && (2.60)\\
\texttt{sb} &= \texttt{s0.replace(" ", "}\text{space}\texttt{")} && (2.61)\\
\texttt{sb.(sb.size()-1)} &= \texttt{sb.(sb.size()-1).replace("...", "}\text{ellipsis}\texttt{")} && (2.62)
\end{aligned}$$

On (2.61), the space characters are all replaced with the `string` literal "space". The `replace()` function does not modify the parent `string`, it instead returns a new one containing the effect of the replacement operation. Like the `split()` function, it also transforms the `string`'s topology in a manner which is shown in Fig. 2.5. Here, the replaced `string` serves as a delimiter such that the new `string` object's elements, all character arrays, stagger between the delimiter ("space" in this example) and the regions between it.

Line (2.62) performs a second replacement operation, this time on the last element of `sb` ("functions..."), wherein the "..." sequence is substituted with the `string` literal "ellipsis". The result, an array of character arrays, is then reassigned to the last element.
As with the `split()`, `replace()` always yields a `string` having a topology as given in Fig.

2.5 – i.e. a sequence of character arrays containing the replaced `strings` – irrespective of the original `string`'s topology.

2.3.9 Running Euclideus Instructions Contained in a string

Generally, seeing as programming languages are defined in text, a `string` object may therefore itself consist of instructions pertaining to that language. This allows programs to be written which can generate computer code – instructions and structure written as text in a given programming language. In the Euclideus language it is possible execute commands contained in a `string` object (as long as they are written in said language) with the help of the `eval()` function, a member of `string`-type objects. A call to a `string`'s `eval()` function tell the interpreter to execute the commands contained in the parent `string` as if they were to be read and executed as proper programming instructions – instructions that are to be interpreted directly as expressions in the syntax of the Euclideus language. The `string`'s `eval()` function takes no arguments.

As a few simple examples, one may execute the following instructions contained in a `string`, by invoking the `eval()` function on the parent `string`:

```
s1 = "array:100;".eval()                                  (2.63)
s2 = ":x=array:100; x.64=i^i, return=x.64;".eval()        (2.64)
s3 = "x=33, y=2/pi".eval()                                (2.65)
                                                          (2.66)
```

In the first example (2.63), a `string` containing an array constructor is contained between quotation marks. The `eval()` function then interprets this statement directly and produces an array of size `100`, which is assigned to `s1`.

In (2.64), the `string` contains a sequence of commands that are enclosed inside a scope bracketed by `:` and `;` therefore any variables created inside that scope are local, except for the `return` variable. Essentially, the instructions contained in (2.64) are equivalent to a function taking no arguments. These instructions are then executed by the interpreter via the `eval()` function, and since a value is given to `return`, this expression thus produces the decimal equivalent of ıı ≈ 0.2079 and assigns it to `s2`. Since `x` is local to the scope enclosed by the `:` and ";" brackets.

In the last example (2.65), a sequence of *unscoped* instructions is contained inside the `string`, in this case any values created upon running this script are persistent since every instruction is executed in the outer scope. Thus, after executing these two commands, `x` carries a value of `33`, `y` carries a value of `2/pi`. Nothing is returned by the `eval()` function, therefore no assignment is made (`s3` carries no value).

3 Class-Based Object-Oriented Programming

Another basic way to create an object is through the use of a class. A class serves two purposes. Firstly, it is a recipe that may be used to create an object. It contains all the necessary information (in the form of a syntax that will be elaborated upon here) that is needed in order to build all of the data and function members of an object. Such a recipe may be inherited from another class, which may then add its own instructions on top of it. As a second purpose, shown in Fig. 1.1, a class is also able to carry its own set of members – these consist of function, object, and class members, termed "static" because they never become instantiated, they remain part of the scope of the class but do not propagate to objects nor to any other (sub) classes that inherit from it.
An object created via a class `A` is said to be an *instance* of `A`, or such an object is of *type* `A`[7]. The following section outlines the basic properties of a class.

3.1 The Class and its Instances

A class is created by assigning a set of assignment instructions to an identifier – a valid class identifier being an expression which contains none of the recognized operators, delimitation symbols or bracketing symbols – prefixed with the `class` keyword. The following is a simple example of a class definition:

```
A = class: u=5, v=-7, f(x)=u+v*x, A(x,y)=:u=x,v=y,return=this; ;     (3.1)
```

As displayed in (3.1), a class called `A` is created by assigning a recipe to it, which is represented on the right-hand side by a set of comma-delimited assignment instructions enclosed by the `:` (opening) and `;` (closing) brackets. This set of assignment commands serves to build a new object by creating and initializing its members. In the above example, (3.1) indicates that an object of of type `A` will be created with two data members, `u` and `v`, of primitive type initialized to values `5` and `-7`, respectively, as well as two functions: one called `f(x)` taking a single argument `x` and returning `u+v*x` and another called `A(x,y)`[8] taking two arguments, `x` and `y`, while not returning any value but merely assigning values to members `u` and `v`. As explained in the procedural reference manual, a function that contains more than one single instruction must use the `:` and `;` opening and closing brackets in order to define its scope, while an algebraic-type function (e.g. `f(x)=x+3`), assigned using a single instruction, does not require these opening and closing brackets. In (3.1), the constructor `A(x,y)` is assigned to a sequence of two instructions (`u=x` followed by `v=y`), and therefore requires these brackets. The final closing bracket on the right-hand side of (3.1) closes the class definition.

An object is built by executing the commands defined on the RHS of (3.1), each command initializing a new object member by assigning a value to it. Note that the scope character-

[7] An object's type can be obtained using the object's `type()` function, which returns a `string` object that represents the object's type.

[8] This function carries the same name as the class. As we will se later, such a method is called a *constructor*, it serves a special purpose which will be detailed later in the section.

istic of the member function `A` in the example above differs slightly from that of a standard function in the global Euclideus environment (i.e. one that does not belong to an object): a standard non-object function is unable to modify the values of any variables existing outside of it, while a function that belongs to an object (often called a *method* in object-oriented terminology) is able to modify all data members belonging to its parent object. Thus for instance, after executing the function `A(10,11*pi)`, members `u` and `v` will carry the values `10` and `11*pi`, respectively.

3.2 Instantiating a Class

In order to create an object as an instance of a class, Euclideus introduces a specific syntax used in order to invoke the set of commands (such as the one given in example (3.1)) that is responsible for building an object. The following presents the simplest manner in which to create an object of type `A` (i.e. of the class `A`):

```
a = A:;
```
(3.2)

In (3.2), an object (here called `a`) is created based on a recipe defined in a class called `A`. This is indicated by the opening `:` and closing `;` brackets immediately following the name of the class of which it is intended to become (here, the class called `A`). The object `a` thus created by executing a set of assignment commands such as those defined in (3.1) will contain two data members `u` and `v`, with values `5` and `-7`, respectively, along with the functions `f(x)` and `A(x,y)`. It is important to note that these four quantities – `u`, `v`, `f(x)` and `A(x,y)` – *are properties of* the object `a`. This may be seen most clearly when invoking those members via an indirection from object `a`, which is provided using the dot `.` operator. Keeping the example (3.1) in mind, the members of object `a` may be accessed as follows:

```
a.u, a.v, a.f(1+y), a.A(x,y),
```
(3.3)

yielding values of `5`, `-7`, `-(2+7*y)`; while the last command[9], as per the instantiation instructions (3.1), returns the object `a` itself as expressed using the "`this`" keyword – which is reserved to represent the parent object – after re-assigning new values to members `u` and `v` – after the last command in (3.3) is executed, `a.u` and `a.v` carry the values `x` and `y`, respectively.

In this example, both members of object `a` are of primitive type, i.e. they are simply algebraic expressions (numbers in this case). Different values may be reassigned to these members, for example `a.u=x` will change the value carried by the member `u` of object `a` to the value of `x` (if `x` doesn't carry any value itself, it is simply taken as a primitive object, an indeterminate variable name in the present example). Since an instance object's members are all defined according to the object's class, they are *immutable*, i.e. they cannot change type. Therefore, reassigning a new value to an instance object's member is only possible *provided that the new*

[9]As we will see, the function `A(x,y)`, whose name is that of the class itself, acts as a constructor for the class.

value is of the same type as the member. Thus, in this example, if `x` were to evaluate to a non-primitive object (an object whose type is not primitive), its type would not match that of `a.u` and therefore the reassignment `a.u=x` would not be possible.

On the other hand, any object that is not a member of an instance object may be reassigned to a different type. Thus, any such stand-alone object is mutable. For example, it would be possible to make the assignment `a=34`, which would change a from type `A` to type primitive. This is an example of a *mutation*, in which an object's type is modified.

It exemplifies the difference between an *immutable* quantity and a *mutable* one. As we will see later, a mutant object's members (in contrast to an instance object's members) are all mutable, meaning that their members may be reassigned to a different type.

3.2.1 The Object Constructor

Keeping example (3.1) in mind, Euclideus provides a second method to create an object from a class, using what is called a *constructor*. Note in (3.1) the presence of a function `A(x,y)` which has the same name as the class `A` to which it is assigned. This particularity signals to the interpreter that `A(x,y)` is to be automatically called upon creation of a new object from the class. In order to make use of the constructor, an instantiation statement must also pass a set of input arguments corresponding to the ones understood by the constructor. Keeping with the present example, the following instantiation will result in an object created by invoking `A(x,y)`:

```
c = A: 5/z, -5 ;                                            (3.4)
```

In (3.4), the arguments used for the constructor are enclosed between the `:` and `;` brackets, as they are not used directly in a function expression, but rather in the instantiation from a class. The new object `c`, again of type `A`, is now created by invoking the function `A(x,y)` behind the scenes during instantiation, using values `5/x` and `-5` for the first and second arguments of the constructor `A(x,y)`, respectively. The newly created object is returned (via the `return=this` statement in the constructor) and stored in `c`. In practice, the "`return=this`" statement may be omitted because a member function having the same name as the class is always interpreted as a constructor, and thus the parent object is always returned.

From the instantiation instructions defined in (3.1), this means that members `c.u` and `c.v` will carry the values `5/z` and `-5`, respectively, upon creation of `c`. It may therefore be verified that a call to `c.f(1/z)` will produce a result equal to `0`.

As a final note, if no constructor is specified in the class definition, Euclideus automatically provides a default constructor `A=:return=this;`, which simply returns the object defined by the instantiation instructions.

3.2.2 Encapsulation

By default, all members of an object are directly accessible from the outside scope, in the sense that they are visible and modifiable from outside of the object. For example, in (3.2) the contents of `a` may be viewed by invoking `a.u` and `a.v`, and moreover they may be modified by assigning new values to them, e.g. the assignments `a.u=21` and `a.v=88+x` will modify these members[10]. In object-oriented applications, it is often desirable to hide, or in other words to *encapsulate* certain parts of an object's composition such that these are only accessible (visible and modifiable) from within the object, not from the outside.

To this end, Euclideus affords a `private` scope that may be included in the definition of a class. Any items (functions, members or inner classes) enclosed within a `private` scope indicate to the interpreter that these items are all accessible (visible and modifiable) only from inside of the object. Consider for example the following class:

```
B = class: u=5, v=-7, f(x)=u+v*x/h, setH(x)=:h=x; getH()=:return=h;   (3.5)
             private:                                                 (3.6)
              h=u/3,                                                  (3.7)
              g(x)=u/h,                                               (3.8)
              B(x,y,z)=:u=x, v=y, h=z+i*z, return=this;               (3.9)
             ;                                                        (3.10)
           ;                                                          (3.11)
```

Lines (3.6-3.10) introduces the `private` scope, any items defined within this scope are deemed *private*, which means that they are inaccessible from outside the instance object. In this example, an instance of `B` will contain private members `h` and `g(x)` in addition to its openly accessible *public* members `u`, `v`, `f(x)` and `B(x,y,z)`, `setH(x)` and `getH()`. Note that the constructor is now called `B(x,y,z)` – its function name must mirror the class name so that it is automatically invoked upon an object's creation using the syntax given in (3.4).

As illustrated on (3.5), the functions `f(x)`, `setH(x)` and `getH()` are able to use or modify the private member `h` owing to the fact that they are all a part of the object's constitution. All members, whether public or private, are accessible *within* the object. However, this is not the case when viewed from the outside scope.

Let's say an object `b` is created from the class defined by (3.5-3.11),

```
b = B: 2, 2, 4*-i ;                                                   (3.12)
```

As may be verified from the class definition, this will result in members `u`, `v` and `h` (of instance `b`) all carrying a value equal to 2. Now although the private members `h` and `g(x)` are a part

[10]An object's functions may never be modified unless the object is a mutant.

of the instance object `b`, any calls to `b.h` or `b.g(x)` will result in an error because they are declared as private members, and are therefore inaccessible from a scope outside of that of `b`. From the outside scope, it's as if `b`'s members `h` and `g(x)` don't exist at all.

Moreover note that the constructor `B(x,y,z)` is also enclosed within the `private` scope, it is therefore also inaccessible from the outside. A function call like `b.B(1,2,3)` is therefore impossible. Thus for any given object of class `B`, this constructor is invoked only once: at the instance's creation using the constructor syntax (3.4). This is because, once created, the resulting object affords no means whatsoever to call its constructor (again because the constructor is located inside a `private` scope). If the public function `setH(x)` – which is the only other one able to modify the value of `h`, in this case from an outside call such as `b.setH(5*t)` – were not implemented, this would imply that the private member `h` would carry the same value for the entirety of `b`'s existence as an object of type `B` [11]. Additionally, the function `getH()` allows to access the private member `h`. Methods like `getH()` and `setH(x)` are known as "getters" and "setters" in object-oriented programming, respectively. Providing one or both of these functions allows to define the level of accessibility – from the outside scope – of a an object's private member. For instance, if `h` is to be visible but not modifiable, then only a getter would be defined for it; while only a setter would be defined should `h` be permanently invisible (but still modifiable).

3.2.3 STATIC MEMBERS

In some cases, certain functionality which is identical between all instances of a given class, or which is logically associated to a class instead of an instance, may be more appropriately included within the class itself instead of reproducing identical functionality and data across all instances. Thus, in addition to providing a recipe for generating objects, a class may also contain its own set of members which are not transferred to instances created from the class. As such members remain in the scope of the class, they are termed *static* members. These `static` members are nonetheless accessible and modifiable from within any instance of the class regardless of their encapsulation level.

`static` members are defined as part of the class by using the `static:;` scope in the class definition. Enclosing quantities inside the `static:;` scope (again between the `:` and `;` brackets) indicates to the interpreter that these are to remain within the class' own scope and never become part of any instance of the class. As an example, consider the following class,

[11] If `b` mutates, it will become of type `mutant`, making all of its members public and mutable. Mutations and mutant-type objects will be elaborated upon later in the present document

```
C = class: u=5, v=-7, f(x)=u+v*x/h, setH(x)=:h=x;            (3.13)
            private:                                          (3.14)
             h=a*u/3,                                         (3.15)
             g(x)=u/h+d*x,                                    (3.16)
             C(x,y,z)=:u=x, v=y, h=z+i*z, return=this;        (3.17)
            ;                                                 (3.18)
            static:                                           (3.19)
             a=e^(i*pi/3),                                    (3.20)
             b=a',                                            (3.21)
             c(x)=(b+a)/x,                                    (3.22)
             private:d=-3;                                    (3.23)
            ;                                                 (3.24)
          ;                                                   (3.25)
```

Here, class C is identical to class B defined through (3.13-3.25), except that it also includes a set of assignments placed inside a `static` scope (3.19-3.24). As a result, members a, b, c(x) and d become properties of the class itself. All statements enclosed within `static:;` are absorbed into the class' scope at the moment it is defined, and the resulting instantiation instructions – understood as the recipe used for creating instances of the class – only include the assignments which are not enclosed within `static:;`. Note also that the private scope also applies to static members such that a class may also have visible and invisible members.

That said, as evidenced by the instantiation instructions (3.7) and (3.8), the `static` members are still fully accessible from within the instances *as if* they were members of the instance.

When accessed by indirection from a class, `static` members are called in a manner which is analogous to instance member indirection calls, but. Because class names and object names are allowed to overlap, a new indirection operator, the double-dot `..` operator, is be defined to call a class' own static members:

```
C..a, C..b, C..c(2),                                          (3.26)
```

will yield values of `0.5+0.866*i`, `0.5-0.866*i` and `0.5` (i.e. $\cos(\pi/3)$), respectively, while a call to `C..d` is impossible because member `d` is enclosed inside a `private:;` scope (it is both private *and* static).

Comparing class B (3.5-3.11) to class B (3.13-3.25), note that instances of either of these classes will look the same: they will contain identical sets of members. However, this doesn't mean that they are equivalent. This may be seen by considering the following circumstances:

```
   D = class:  u=6, v=C:; ;                                      (3.27)
   d = D:;                                                       (3.28)
   b = B:;                                                       (3.29)
 d.v = b, ⟵ this assignment is impossible                        (3.30)
   b = d.v ⟵ this assignment is possible                         (3.31)
```

Since classes B and C yield objects having the same structure (recall that the only difference between these classes is that C contains static members while class B does not), the instance member d.v and the object b ostensibly have the same exact structure (their members and functions are all identical). It would seem possible to reassign d.v to a value equal to b. However, because these two objects are of different types (C and B, respectively) and v is a member of instance d, it is immutable and may not be reassigned to a different type. In contrast, b does not belong to another object (it is not an instance member of another object) and so therefore it is mutable, allowing for it to be assigned to a value of a different type. Thus, assignment (3.31) mutates object b from type B to type C and copies into it all members of d.v.

3.2.4 The Object Destructor

In analogy to the object constructor, Euclideus also provides syntax for implementing a destructor, a function that is called right before an object is deleted. The destructor is an instance function taking no arguments, returning no value and whose name is the class' name prefixed with an underscore (_). The following is an example implementation of a class with a destructor:

```
DD = class:                                                            (3.32)
      private:                                                         (3.33)
         u=3, DD(x)=:u=m+1/x, return=this; _DD()=:m=0; setM()=:m=u;    (3.34)
      ;                                                                (3.35)
      static:  private:m=0; ;                                          (3.36)
     ;                                                                 (3.37)
```

Here, the class DD contains a single static member m initialized to 0 when the class is created. When an instance dd=DD:v; is created from this class, the constructor DD(x) is invoked and initializes dd.u to a value m+1/v. The instance dd also contains a function setM() which assigns the current value of dd.u to the static member m (a member of the class DD). Thus, m happens to be completely volatile because it may be modified by any instance created from DD. When the instance is destroyed, e.g. via a call to delete(dd), the destructor _DD() is called right before the object's destruction, which in this example resets m's value to 0.

3.2.5 Two Examples of Singleton Classes

To synthesize the above concepts, this section provides two example implementations of singleton classes. The singleton is an often-used design pattern in which a class can be instantiated only once, i.e. only a single instance object can be created.

One way to satisfy this pattern is to define the singleton class with a *static* instance member, as in the class `singleton1` shown in Fig. 3.1-a. I this case the singleton instance exists within the class' scope and thus the instance is created with the class itself. Instantiation of the class (lines 26-28), e.g. via `s1 = singleton1:;`, is trivial – it merely returns an object `s1a` of type `singleton1`. `s1a` contains no members, since none are defined in `singleton1`'s instantiation instructions (apart from the trivial default constructor). Getter and setter functions to the singleton's value (`val`) are respectively provided by `getVal()` and `setVal()`, which are also `static`, therefore part of the class.

Note that an arbitrary number of `singleton1`-type objects bay be created via instantiation, however each will be associated to the same `static` value `val`, which is created only once, inside the `singleton1` class. For example, if a second object `s1b` is created and modifies `val`, i.e.

$$\texttt{s1b=singleton1:;} \tag{3.38}$$

$$\texttt{s1b.setVal(newVal)} \tag{3.39}$$

then the change will affect `s1a` (and any other `singleton1` instance): `s1a.getVal()` will now return `newVal`, again because the singleton pattern ensures that only a single copy of `val` exists.

Another more explicit implementation of the singleton pattern is given by the class `singleton2` defined in Fig.3.1-b. In this case, the instantiation of the singleton is explicitly restricted to a single instance via a `static` flag `n`, which represents the number of instances of `singleton2`, contained in the `singleton2` class, while the the remaining data and function members are delegated to the instance object, not the class.

Of particular importance in this implementation is the structure of the constructor defined on lines 20-25. Note that the `return=this` statement is enclosed inside an `if:;` block, which means that the constructor only returns a `singleton2` instance if `n` is `0`, which case it increments `n` to `1`. Thus, the instantiation of `singleton2` ensures that no additional instances may be created, that is, unless the existing one is destroyed, in which case the destructor (lines 27-29) ensures that `n` is decremented to zero.

3.3 Inheritance

Classes have the ability to inherit the full set of instantiation instructions of other classes, in addition to defining their own set of instantiation characteristics. The class inheriting proper-

```
singleton1=class:

  static:

    setVal(x)=:
      val=x
    ;

    getVal()=:
      return=val
    ;

    private:
      val=1
    ;
  ;

  private:

    singleton1()=:
      return=this
    ;
  ;
;
```

(a)

```
singleton2=class:

  static:
    private:
      n=0;
  ;

  setVal(x)=:
    val=x
  ;

  getVal()=:
    return=val
  ;

  private:

    val=1,

    singleton2()=:
      if: ~n,
        n=1,
        return=this
      ;
    ;

    _singleton2()=:
      n=0
    ;
  ;
;
```

(b)

Figure 3.1: Two example implementations of singleton classes (a) by defining the singleton object as a `static` member of the class, and (b) by using a `static` flag (n) that controls instantiation.

ties is referred to as a *subclass* while the classes from which it inherits are called *superclasses.* Inheritance is limited to instantiation instructions, classes never inherit static members of their superclasses.

3.3.1 Simple Inheritance

As an example of inheritance, consider the following simple example:

```
SuperC=class:                                              (3.40)
    u=3,f(x)=1/x,                                          (3.41)
    private:                                               (3.42)
        v=-3,                                              (3.43)
        SuperC(x,y)=:u=x,v=y,return=this;                  (3.44)
    ;                                                      (3.45)
;                                                          (3.46)
SubC:SuperC;=class:                                        (3.47)
    u=e,f(x)=e^x,                                          (3.48)
    private:                                               (3.49)
        SubC(x,y,z)=:u=x,SuperC_SuperC(y,z),return=this;   (3.50)
    ;                                                      (3.51)
;                                                          (3.52)
```

Lines (3.40-3.46) define a class `SuperC` containing two public members `u` and `f(x)` as well as a private member `v` and a private constructor `SuperC(x,y)` which serves to initialize members `u` and `v` upon instantiation. The next lines (3.47-3.52) define a class `SubC` which inherits the properties of `SuperC`.

Notice that the left-hand side (LHS) of (3.47) is an expression containing the name of the subclass class with the superclass enclosed between `:` and `;` brackets. This particular syntax is to indicate to the interpreter that the new (here, `SubC`) class will inherit all instantiation instructions from the one enclosed between the brackets. Furthermore, whatever lies on the RHS of the assignment statement consists of instantiation instructions belonging specifically to the subclass `SubC`, which are added to the set of instantiation instructions inherited from `SuperC`. The encapsulation attribute of each inherited member is preserved in the subclass, thus private members remain private and public ones remain public in the subclass. Note also that the subclass `SubC` defines its own private constructor `SubC(x,y,z)` which initializes member `u` and invokes another function `SuperC_SuperC(y,z)`.

Now, note that the superclass `SuperC` and the subclass `SubC` prescribe members – `u` and `f(x)` – with overlapping names. An immediate question therefore arises: Which of these members will an instance of `SubC` contain – those from `SuperC` or those from `SubC`? This question may be resolved by modifying the names of all instantiation members of the superclass upon their appropriation by the subclass.

In Euclideus, this is accomplished by simply prefixing the superclass' name to the name of each inherited member. Thus, `u` from class `SuperC` becomes `SuperC_u` once inherited by class `SubC`, member `f(x)` from the superclass becomes `SuperC_f(x)`, and so on. The members of the subclass all retain their original names as defined by the RHS of (3.64).

As a result of (3.40-3.52), the subclass would create instance objects according to a prescription in the vein of

```
SubC = class: u=e, f(x)=e^x,                                      (3.53)
              SuperC_u=3, SuperC_f(x)=1/x,                        (3.54)
              private:                                            (3.55)
                SuperC_v=-3,                                      (3.56)
                SuperC_SuperC(x,y)=:u=x, v=y, return=this;        (3.57)
                SubC(x,y,z)=:u=x, SuperC_SuperC(y,z), return=this; (3.58)
              ;                                                   (3.59)
            ;                                                     (3.60)
```

Here, (3.58) clarifies the action of the constructor `SubC`. As all members of the superclass (including its constructors) are inherited and prefixed with "`SuperC_`", the subclass' constructor is actually invoking the the superclass constructor, which is called `SuperC_SuperC(x,y)` once it is inherited. Implemented in this manner, in this manner, the subclass constructor `SubC(x,y,z)` is able to initialize all members (including inherited ones) upon instantiation. To see this more clearly, let us instantiate the class `SubC` using its constructor:

```
subInstance = SubC:a,b,c;                                         (3.61)
```

The instructions (3.53-3.60) produce an instance (3.61) of `SubC` which contains all of the prescribed members, among which `subInstance.u`, `subInstance.SuperC_u` and `subInstance.SuperC_v` are initialized to values `a`, `b` and `c` by the constructor `SubC(x,y,z)`, respectively.

3.3.2 MULTIPLE INHERITANCE

The previous example outlined the basic mechanics of a class inheriting from another one. In general, classes are allowed to inherit properties from any number of previously defined classes. This is achieved simply by listing all the superclasses in between the opening and closing (`:` and `;`) brackets on the left-hand side of the subclass definition.
This is illustrated below through an example of a class `C` inheriting from both classes `A` and `B`:

```
      A = class:  u=3, v=-3/pi, w(x)=x^(v+u) ;                            (3.62)
      B = class:  u=e, v=3/e, w(x)=x*u-v ;                                (3.63)
C:A,B; = class:  u=44, v=3/4, w(t)=1/(u*v-pi*t), static:w=-9; ;         (3.64)
```

Notice that the left-hand side (LHS) of (3.64) is an expression containing a set of names (A and B), separated by a comma and enclosed by the usual : and ; brackets. In the above example, the subclass is therefore C and its superclasses are A and B. C thus inherits *all* instantiation properties from both A and B. Consequently, all instances of C will also be instantiated with all properties from A and B.

Again, since the superclasses A and B prescribe members with overlapping names, which also overlap with those of class C, the interpreter prefixes their members with A_ and B_, respectively, before incorporating them into class C.

As a result of (3.62-3.64), the subclass C would create instance objects according to a prescription such as

```
C = class: u=44, v=3/e, w(t)=1/(x-pi*t),                        (3.65)
           A_u=3, A_v=-3/pi, A_w(x)=x^(A_v+A_u),                (3.66)
           B_u=44, B_v=3/4, B_w(t)=1/(B_u*b_v-pi*t) ;           (3.67)
```

The subclass again appropriates all of its superclass members while renaming them by prefixing them with the name of their own class, and defines its own set of members according to the instantiation instructions given on the RHS of (3.64). It is important to note that class inheritance does not imply any information sharing between subclass and superclass, nor between instances thereof. Even if a class inherits from another, it nevertheless stands on its own as a completely distinct class. Objects created from such a subclass do not share any members with any of its superclasses.

Adhering to this kind of naming convention ensures that an object member's heredity line remains obvious from its name. If, say, another class D were to inherit from C, all quantities in D's instantiation instructions originating from C would be in turn prefixed with "C_", i.e. D would contain members called C_A_U, C_B_u, C_A_w(x) and so on. It must be specified that an object created from class C, as per the example above, will *only* be recognized as being of type C, it will not be recognized as an instance any of the superclasses it inherits its properties from.

Lastly, note that (3.64) also defines a static member w belonging to class C (invoked using the syntax C..w). Naturally, w will therefore not be part of any object instantiated from C nor will it be transferred to any other class that inherits from C; it will remain within the scope of C during the entire lifetime of the class.

4 Prototype-Based Programming

As detailed in the previous section, the flexibility afforded by mutable members makes it possible to easily create more complex structures. While classes allow for the composition of a great variety of different objects, by their very nature these instances have a fixed structure as prescribed by their class, their members are not mutable. In order to create a different structure, a new class must be defined either from scratch or via inheritance.

4.1 The mutant Object

To reduce the rigidity of class-based objects, Euclideus provides a second object-oriented paradigm, known as prototype-based programming, which combines the ability to create general object structures as illustrated in Fig. 1.1 with the flexibility afforded by mutations. This kind of object, of type `mutant`, is entirely mutable – all of its members are mutable (their type can be changed via reassignment) and members may be spontaneously added to or deleted from it. In addition, a mutant object does not encapsulate any of its members; all members of a mutant are public.

4.2 Mutating a primitive Object

Euclideus currently offers two ways to create a mutant. The simplest is to mutate a `primitive` object by adding members to it, as in the following example:

$$\texttt{p = 3} \tag{4.1}$$
$$\texttt{p.u = 4} \tag{4.2}$$
$$\texttt{p.v = pi/2} \tag{4.3}$$
$$\texttt{p.f(x) = v/x} \tag{4.4}$$

On line (4.1), a `primitive` type object is first created by assigning a value of `3` to `p`. The assignment on line (4.2) *adds a new member* to `p`, thus mutating it from type `primitive` to type `mutant`. Upon mutating from a `primitive` type, `p` still carries its original value along with it in a member called `p.val` which contains the original value of `3` – such that a call to `p.val` returns a value of `3` – along with its new member `p.u` carrying a value of `4`. Lines (4.3) and (4.4) then add a new data member `p.v` as well as a new function member `p.f(x)`. Keeping in mind that mutants offer no encapsulation, all members of `p` are publicly accessible and modifiable, including any function members.

Having created a basic object from a primitive, an entire set of clones may be spawned by copying object `p`. As such, `p` is taken as a *prototype.* The prototype essentially serves a role analogous to that of a class; though it only *proposes* an initial structure for its clones, unlike the class it does not enforce it. Any of its clones may in turn serve as a prototype for other sets of mutants. The following example creates an array of one thousand clones of the prototype object p:

```
clones = array:1000;                                      (4.5)
for:j=0, j<clones.size(), j=j+1, clones.j=p;              (4.6)
```

An array called clones, first created on line (4.5), is then passed through a loop (line (4.6)) which assigns p to each of its elements, thus creating 1000 identical copies of the prototype object p.

Aside from adding members, a mutant also allows for removal of its members. The global delete() function may be used to this end. After effectuating the following deletions,

```
delete(p.u, p.f(x))                                       (4.7)
```

only two members remain: p.v and p.val. Note that this deletion operation does not affect the clones (4.5-4.6) which were already obtained from a previous version of p.

4.3 Mutating an Instance Object

For any given class, instances of this class may be directly mutated either via the global mutate() function or – as a brute-force approach – by deleting their class (via the delete() command), in which case all instances of the deleted class automatically become mutants. Consequently,

1. Upon becoming a mutant, (i) the object no longer is associated to the deleted class (its type changes to mutant), (ii) all its members become public, mutable – they can all be reassigned to a different type, and may also be turned into mutants themselves via the mutate() function – and (iii) the object's structure becomes entirely editable – any of its members can be deleted (using the delete() function) or added to the mutant by simply assigning a new member to it.
2. An object that contains a member which is an instance of the deleted class will also become a mutant itself according to the procedure outlined in 1. In other words, instance objects cannot contain members of type mutant.

As a result, all objects that become mutants upon deletion of a class will immediately contain all of their original data and function members.

The following example is meant to elucidate the above concepts. First, let classes A , B, C and D be defined as follows:

```
     A = class: u=6, private:A(x,y)=:u=x, v=y; f(x)=u*v*sin(x), v=7; ;
                                                                   (4.8)
     B = class: u=31, g(x,y)=x^y, private:  v=A:4/t,t^2; ;         (4.9)
                static:w=sqrt(-9); ;                               (4.10)
   C:A; = class: u=-45, v=pi/u, private:w(x,y)=36^(x+i*y); ;       (4.11)
 D:A,B; = class: u=-45, private:v=C:; w(x,y)=36^(x+i*y); ;         (4.12)
```

The set of instantiation instructions of class A, as defined on (4.8), contains a public primitive data member u, and three private members – a constructor A(x,y), a private function f(x) and a private data member v. For class B, defined on (4.9-4.10), instantiation instructions define public members u and g(x,y) and a private member v of type A initialized using class A's constructor, along with its own private constructor B(x,y). Line (4.10) also defines a static member w which remains as part of the scope of class B.

Lines (4.11) and (4.12) define two other classes C and D, with C inheriting only from A, and D inheriting from both A and B. As a result, instances of D will contain more members than instances of C.

Now, let us create objects by instantiating these four classes:

```
a=A:; b=B:; c=C:; d=D:;                    (4.13)
```

with a, b and c and d being of type A, B, C and D, respectively. According to the classes defined in (4.8-4.12), a and c contain no dependencies on other classes apart from their own (classes A and C, respectively). Although c is an instance of a class inheriting from A, none of c's members contain dependencies on other classes. On the other hand, b and d each contain a member that depends on class A – both b.v and d.B_v are of type A.

As a first case, let's now consider the deletion of class B by simply invoking the delete operation, delete(B). Firstly, the class' static member w is automatically deleted with the class. Since only b depends on class B (by virtue of directly being an instance of it), b's type changes to mutant. As a result, all of its members become public, mutable and members can all removed or added to it. Note also that the constructor B(x,y) naturally disappears, as such instances may no longer be created once the class is deleted.

As an example, consider the following modifications to the mutant b:

```
         b.u = C:;                          (4.14)
         delete(b.g(x,y)),                  (4.15)
         b.v = D:;                          (4.16)
         b.w = A:-3,4;                      (4.17)
    b.f(x,y) = x/(1+y^2),                   (4.18)
         mutate(b.u)                        (4.19)
```

After executing (4.14-4.19), member function b.g(x,y) no longer exists, members b.u and b.v are reassigned to different types (C and D, respectively), and two new members are added to b – w of type A and a function f(x,y). Also, note the usage of the mutate() function on line (4.19), which transforms b.u from a type C object (as per (4.14)) into a mutant type. The mutate() function transforms an instance object into a mutant; it obviously has no effect on members of an instance object. After (4.19), the mutant b therefore contains a mutant member (b.u), a member of type D (b.v) and a member of type A (b.w), along with a lone function member `b.f(x,y)`.

Now, with `a`, `c` and `d` being the remaining non-mutants, let's now consider the effects of deleting class `A`. This operation affects two of the objects. First, `a` becomes a mutant because it is a direct instance of `A` (this situation mirrors the previously described mutation of `b`).

As for `d`, though the right-hand side of (4.12) does not display overt dependence on `A`, note that `D` inherits properties from `B`, whose own instantiation instructions do prescribe inheritance of properties from `A` (through member `v`). In class `D` this member, of type `A`, is called `B_v` by virtue of it being inherited from (the now deleted) class `B`. Through this indirect dependence, instance `d` therefore also becomes a mutant along with its member `d.B_v` upon deletion of class `A`. All the other members of `d` retain their type, however they become both public and mutable.

Although `b` was already a mutant before class `A` was removed, this does not imply that `b` is unaffected by the deletion of `A` seeing as two of its members – `b.v` and `b.w`, respectively of types `D` and `A` – are dependent on `A`. These two members therefore become mutants leaving `b` with no remaining non-mutant members.
Finally, because none of `c`'s members contain any `A`-dependence, `c` remains unaffected.

5 Future Additions

For the short term, the Euclideus language will see the inclusion of a larger variety of native objects – namely lists, associative arrays and a native matrix type – each with their own specific indirection syntax. In addition to core optimizations, future versions will also contain additional capabilities such as "by-reference" argument passing in functions, mutlithreading, encryption and a proper I/O system.

6 Acknowledgments

The author gratefully acknowledges Peter Renkel for valuable discussions and contributing fundamental ideas.